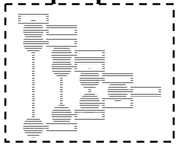
Information Sciences & Technology
Ames Research Center

Generalized Symbolic Execution for Model Checking and Testing

Input plan



complex input structure

Rover Executive

```
void Executive::
startExecutive(){
 runThreads(); …}
void Executive::
executePlan(…) {
 while(!empty)
 executeCurrentPlanNode();
} …
```

concurrency, dynamic data (lists, trees)

execute action

environment/ rover status

large environment data

Future mission software:
- concurrent
- complex, dynamically allocated data structures (e.g., lists or trees)
- highly interactive:
  - with complex inputs
  - large environment
- should be **extremely reliable**

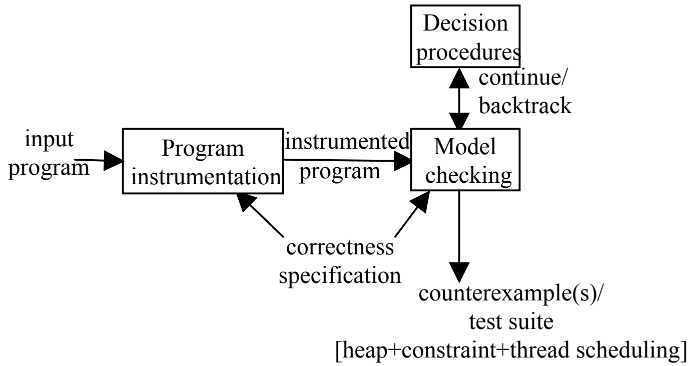
Current practice in checking complex software:

- testing:
- requires manual input
- typically done for a few nominal input cases
- not good at finding concurrency bugs
- not good at dealing with complex data structures

- model checking:
- automatic, good at finding concurrency bugs
- not good at dealing with complex data structures
- feasible only with a small environment
  - and a small set of input values

Our **novel symbolic execution framework**:
- extends model checking to programs that have complex inputs with unbounded (very large) data
- automates test input generation

Framework:



- modular architecture: can use different model checkers/decision procedures

Code

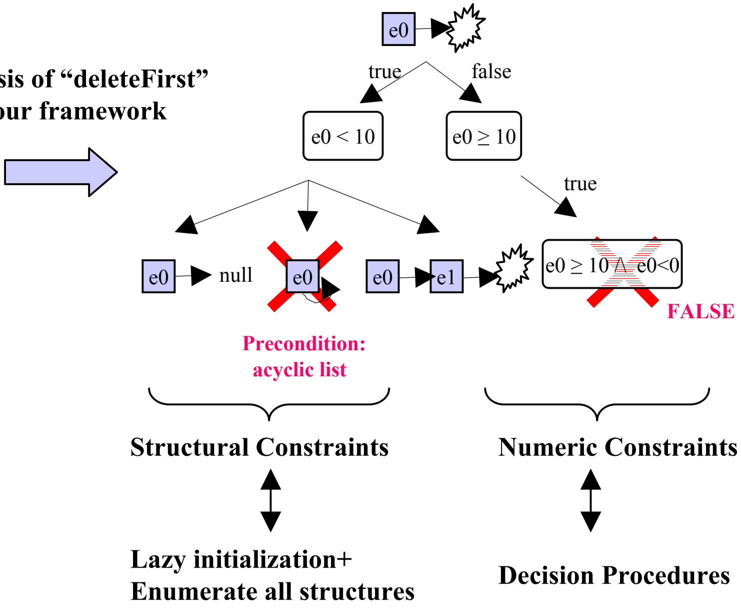
```
class Node {
  int elem;
  Node next;

 Node deleteFirst() {
  if (elem < 10)
    return next;
  else if (elem < 0)
    assert(false);
  … } }
```

-"simulate" the code using **symbolic values** instead of program data; **enumerate** the input structures lazily

(✸ = "unknown yet")

Analysis of "deleteFirst" with our framework



Explanation of Accomplishment

- **POC:** Corina Pasareanu (Kestrel Technology LLC) and Willem Visser (RIACS/USRA)
- Joint work with Sarfraz Khurshid (MIT)
- **Background:** Future mission software systems, which will be concurrent and will manipulate complex dynamically allocated data structures, should be extremely reliable. These kinds of systems are known to be hard to test. Even using model checking to discover errors is inadequate due to the large and complex input domains of such systems.
- **Accomplishment:** A novel symbolic execution framework was developed and an algorithm implemented that enables model checking of programs that have complex inputs with unbounded (very large) data. The algorithm also automates test input generation.The symbolic execution algorithm has been implemented in the Java PathFinder model checker toolset. It has been used to generate test inputs for code coverage (i.e., condition coverage) of an Altitude Switch used in flight control software.
- **Future Plans:** We intend to apply our framework to the analysis of other complex systems, including the Mars K9 Executive prototype. We plan to investigate the application of abstraction techniques in the context of our framework.
- **Overview of Algorithm:** We provide a two-fold generalization of traditional symbolic execution methods. One, we define a source to source translation to instrument a program, which enables standard model checkers to perform symbolic execution of the program. The program instrumentation enables a model checker to automatically explore different configurations of the input data structures and manipulate logical formulae on program numeric values (using a decision procedure). Two, we give a novel symbolic execution algorithm that handles dynamically allocated structures (e.g., lists and trees), primitive data (e.g., integers and strings) and concurrency. To symbolically execute a program, the algorithm uses lazy initialization, i.e., it initializes the components of the program's inputs on an ``as-needed'' basis, without requiring an a priori bound on input sizes. The algorithm uses preconditions to initialize inputs only with valid values.